
UNIZG-FER 222464

Web Architecture, Protocols, and Services

# Web Notification (Web Push) Techniques

## Asynchronous Web Protocols and Browser Networking APIs

- XMLHttpRequest (AJAX)
- Server-Sent Events (SSE)
- WebSocket

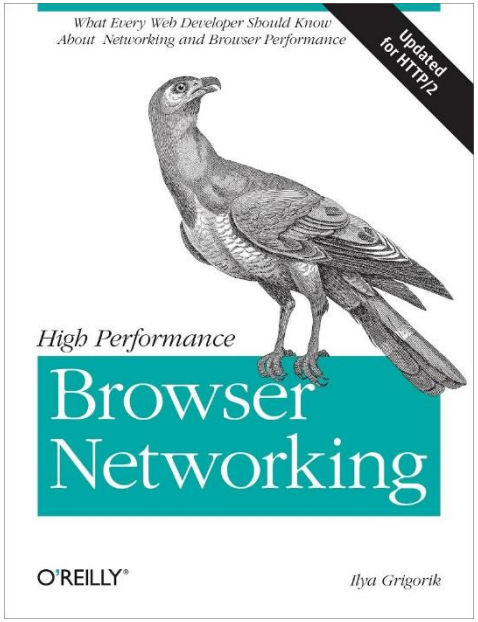
# Reading Material

- **High Performance Browser Networking**

  Ilya Grigorik

  O'Reilly Media, September 2013 (total pages: 383)

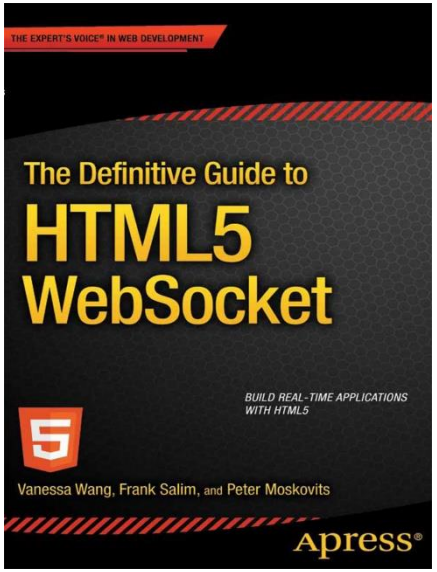
  Relevant content: **Part IV: Browser APIs and Protocols**




- **The Definitive Guide to HTML5 WebSocket**

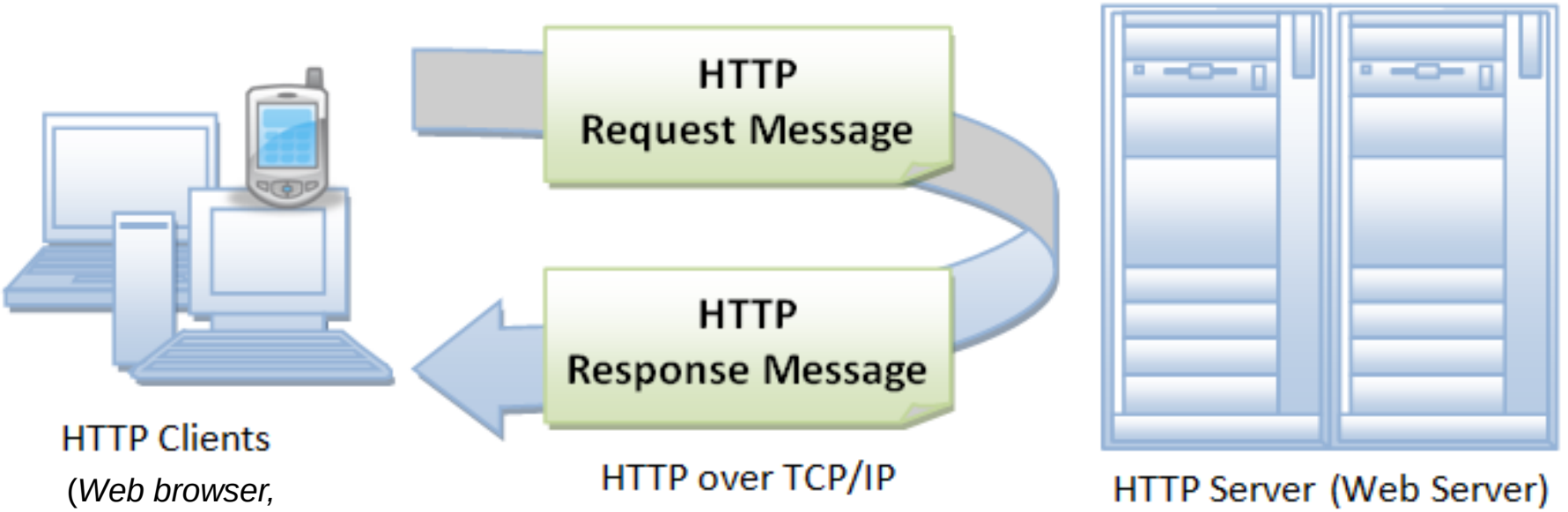
  Vanessa Wang, Frank Salim, Peter Moskovits

  Apress, 2013 (total pages: 188)



# HTTP Limitations

- Synchronous protocol
  - Strictly follows the request-response communication pattern
    - Server never sends data to the client without explicitly being asked for that data by the client



# HTTP Limitations


- Satisfactory for applications the protocol was initially designed for

- Modern web applications require communication patterns HTTP protocol lacks the support for

  - Asynchronous data transfers
    - Examples: sensory readings
      stock exchange monitoring
      live sports results
  - Data streams
    - Continuous data delivery

**Real-time notification**

- TCP/IP connection
  - Bidirectional and asynchronous
- HTTP over TCP/IP
  - ***Artificially*** limited to bidirectional, but synchronous mode

# Web Application Development Limitations

- Plain old web browser
  - Web application developer has little or no control over how and when an HTTP request would be dispatched
  - End user actions
    - **Entering URL in browser's address bar**
    - Clicking an active link
    - Submitting a form

# Web Application Development Limita

- End user actions
  - Entering URL in browser's address bar
  - **Clicking an active link**
  - Submitting a form

```
<html>

 <head></head>

 <body>
  <h1>Title</h1>
  <h2>Subtitle</h2>
  <p>Text paragraph</p>
  <img src="http://www.c.com/image.gif" />
  <a href="http://www.b.com/page2.html">Another page</a>
  <form method="POST" action="http://api.sms.com/sendsms">
   <input type="text" name="phonenumber" />
   <input type="text" name="msgtext" />
   <input type="submit" />
  </form>
 </body>

</html>
```

# Web Application Development Limita

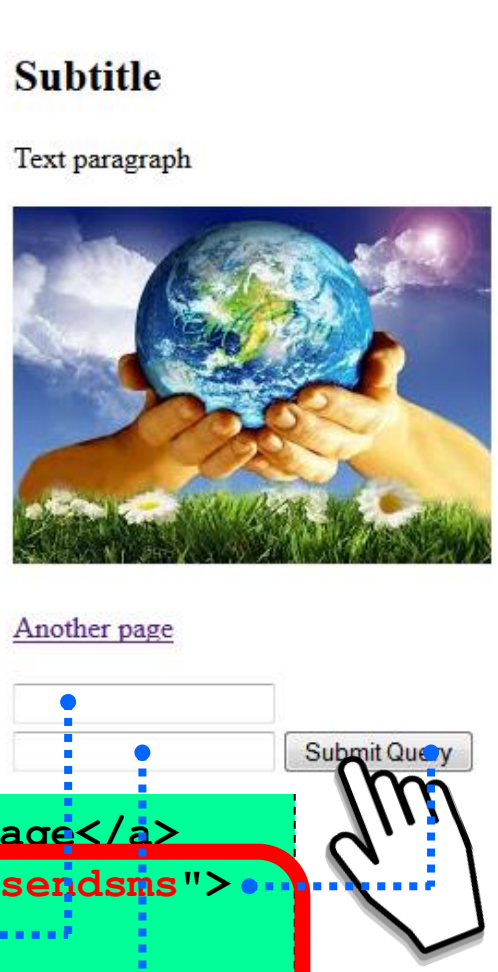
- End user actions
  - Entering URL in browser's address bar
  - Clicking an active link
  - **Submitting a form**

```
<html>

 <head></head>

 <body>
  <h1>Title</h1>
  <h2>Subtitle</h2>
  <p>Text paragraph</p>
  <img src="http://www.c.com/image.gif" />
  <a href="http://www.b.com/page2.html">Another page</a>
  <form method="POST" action="http://api.sms.com/sendsms">
   <input type="text" name="phonenumber" />
   <input type="text" name="msgtext" />
   <input type="submit" />
  </form>
 </body>

</html>
```



# Web Application Development Limita

- Automatic browser-initiated actions
  - Fetching of embedded subresources

```
<html>

 <head></head>

 <body>
  <h1>Title</h1>
  <h2>Subtitle</h2>
  <p>Text paragraph</p>
  <img src="http://www.c.com/image.gif" />
  <a href="http://www.b.com/page2.html">Another page</a>
  <form method="POST" action="http://api.sms.com/sendsms">
   <input type="text" name="phonenumber" />
   <input type="text" name="msgtext" />
   <input type="submit" />
  </form>
 </body>

</html>
```

# Web Application Development Limita

- Limited application-specific control
  - Run-time change of subresource URL
  - Resource gets reloaded from server

```
<html>

 <head></head>

 <body>
  . . . . .
  <img id="myImage"
       src="http://www.c.com/globe.gif" />
  . . . . .
  <script>
    function changeImage() {
      document.getElementById("myImage").src =
                              http://www.c.com/sunflower.jpg
    }
  </script>
 </body>

</html>
```


UNIZG-FER 222464

Web Architecture, Protocols, and Services

# XMLHttpRequest (XHR)
# (Asynchronous JavaScript And XML)

# XMLHttpRequest (XHR)


- Browser-level API that enables the client to script data transfers via JavaScript
  - XHR made its first debut in Internet Explorer 5 in 1999
  - One of the key technologies behind the **Asynchronous JavaScript and XML (AJAX)** revolution
  - XHR is now a fundamental building block of nearly every modern web application
- Prior to XHR, the web page had to be refreshed to send or fetch any state updates between the client and server
- With XHR, this workflow could be done asynchronously and under full control of the application JavaScript code

# XMLHttpRequest (XHR)


- Common XHR usage pattern

```
var xhr = new XMLHttpRequest();
```

- *instantiates new XHR object to use in web application*

```
xhr.open('GET', '/image.jpg');
```

```
xhr.open('GET', 'http://thirdparty.com/image.jpg');
```

- *initializes new HTTP request*

```
xhr.onload = function() {

   .....

};
```

- *callback function invoked automatically by the browser once the HTTP response from the server has arrived*

```
xhr.send();
```

- *sends HTTP request to the server*

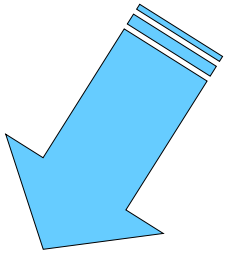
# XMLHttpRequest (XHR)

- XHR dynamics

```
xhr.send();
```

**HTTP request**

```
GET /image.jpg HTTP/1.1
Host: thirdparty.com
```



**HTTP response**

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 9876

[image data representation]
```

```
xhr.onload = function() {
   .....
   this.status
   this.response
   .....
};
```

# XMLHttpRequest (XHR)


- XHR can transfer both text-based and binary data (*not limited to XML as the name may suggest*)

- The browser offers automatic encoding and decoding for a variety of native data types

| Native Data Type | Description |
|---|---|
| `Text` | A simple text string |
| `Document` | Parsed HTML or XML document |
| `JSON` | JavaScript object representing a data structure defined using JSON |
| `ArrayBuffer` | Fixed-length binary data buffer |
| `Blob` | Binary large object of immutable data |

# Example: Downloading Data with XHR


```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/socialdata/friends_online');

/* By default, the browser relies on the HTTP content-type
   negotiation to infer the appropriate data type (e.g.,
   decode an application/js
   Otherwise, the applicati
   data type when initiatin
xhr.responseType = 'json';

xhr.onload = function() {
  if (this.status == 200) {
    for (i=0; i < this.resp
      var friend = document
      friend.innerText = th
      friend.href = this.re
      document.body.appendC
    }
  }
};

xhr.send();
```

**HTTP request**

```
GET /socialdata/friends_online HTTP/1.1
Host: example.com
```

**HTTP response**

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1234

[
  { "name": "John Smith",
    "profile_page": "http://..." },
  .....
  .....
]
```

# Example: Uploading Data with XHR

```
/* Uploading simple textual data */
var xhr = new XMLHttpRequest();
xhr.open('POST', '/upload');
xhr.onload = function() { ... };
xhr.send('This is my text.');
```

**HTTP request**

```
POST /upload HTTP/1.1
Host: example.com
Content-Type: text/plain
Content-Length: 16

This is my text.
```

```
/* Uploading form data */
var formData = new FormData();
formData.append('id', 123456);
formData.append('topic', 'performance');
var xhr = new XMLHttpRequest();
xhr.open('POST', '/upload');
xhr.onload = function() { ... };
xhr.send(formData);
```

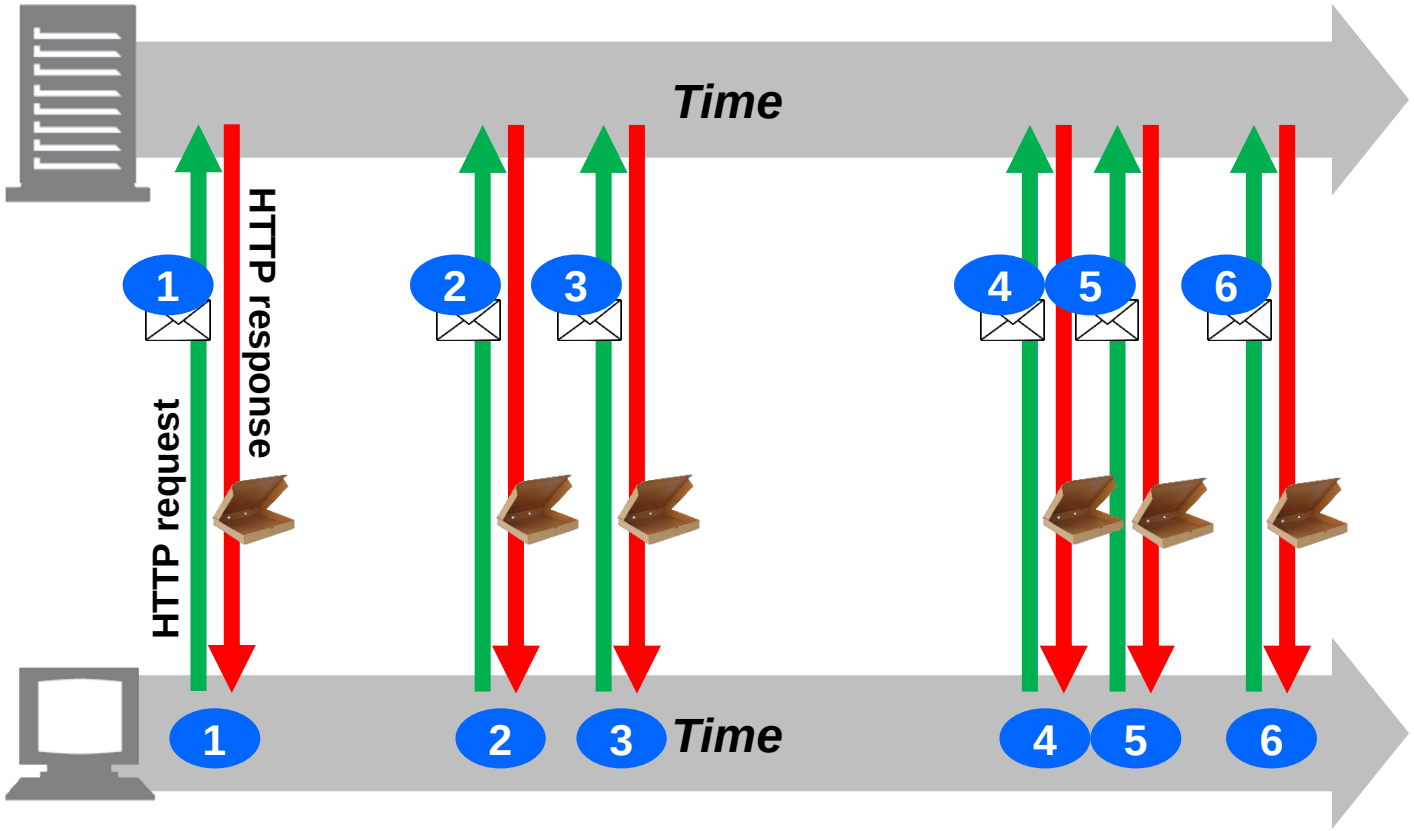
**HTTP request**

```
POST /upload HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

id=123456&topic=performance
```

# Real-Time Notifications on the Web

- Client-To-Server notifications
  - How to synchronize client updates with the server?
  - Client sends an HTTP request with notification data in message body
  - Server responds with empty HTTP response (headers only, no body)



# Real-Time Notifications on the Web


- Client-To-Server notifications
  - Implementation using *XMLHttpRequest*
  - Each notification is implemented as a separate **XHR upload pattern**

```
var xhr = new XMLHttpRequest();

xhr.open('POST', '/update');

xhr.onload = function() {
  if (this.status != 200 and this.status != 204) {
    document.body.write('Update operation failed');
  }
};

xhr.send(data);
```

# Real-Time Notifications on the Web


- Client-To-Server notifications

Example

- Size of notification payload data: 10 bytes
- Typical size of HTTP request header: cca. 500 bytes
- Typical size of HTTP response header: cca. 500 bytes

$$\boldsymbol{efficiency\ of\ communication} =$$

$$= \frac{size\ of\ payload\ data}{HTTP\ request\ size + HTTP\ response\ size}$$

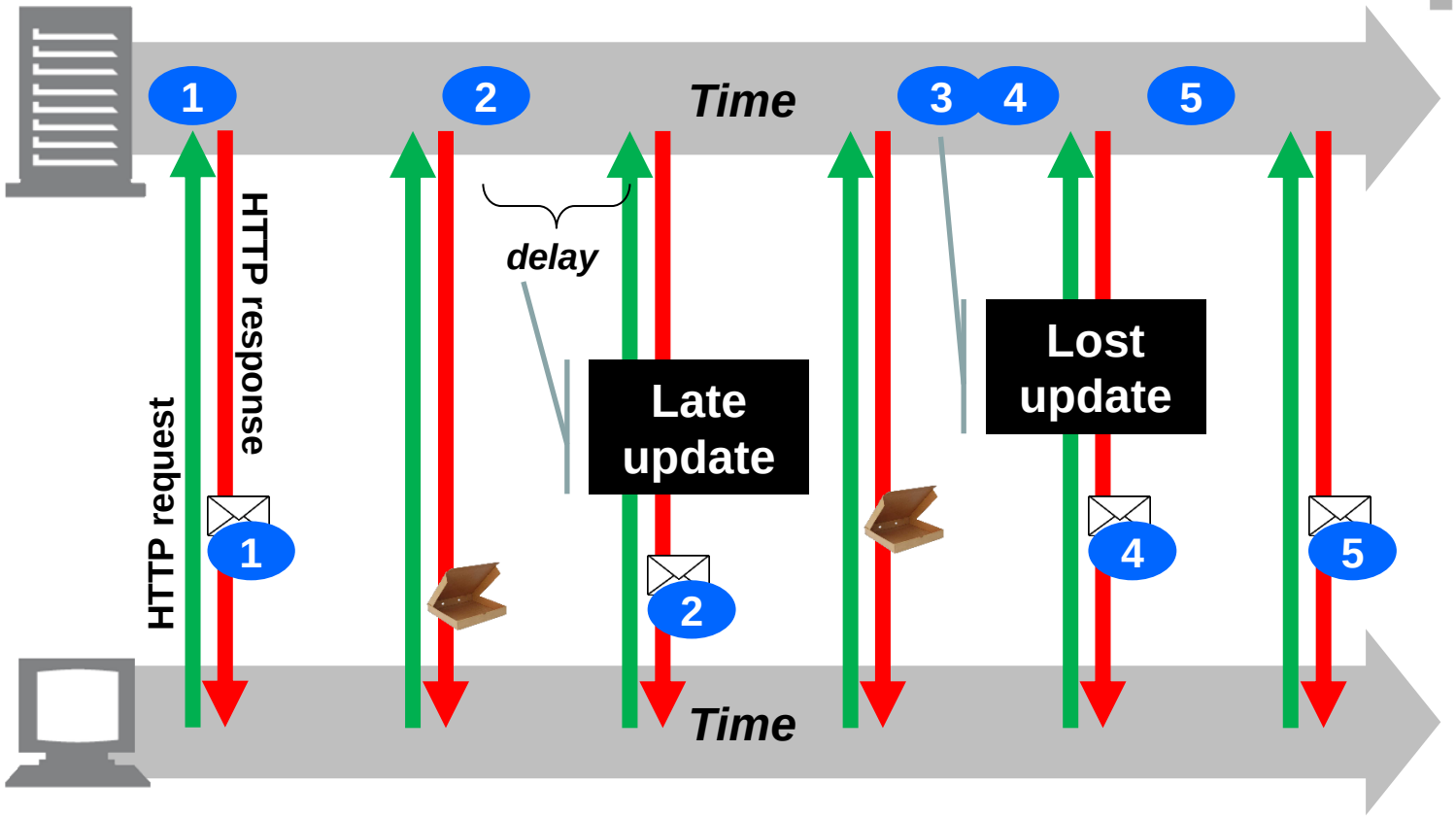
$$= \frac{payload\ size}{request\ header\ size + request\ body\ size + response\ header\ size}$$

$$= \frac{payload\ size}{request\ header\ size + payload\ size + response\ header\ size}$$

$$= \frac{\mathbf{10}}{\mathbf{500 + 10 + 500}} \approx \mathbf{1\%}$$

# Real-Time Notifications on the Web

- Server-To-Client notifications
  - How to synchronize server updates with the client(s)?
  - In HTTP, server cannot deliver any data to the client, without being explicitly asked for them
  - Simplest solution: clients need to do periodic checks ⇒ **POLLING**



# Real-Time Notifications on the Web


- Server-To-Client notifications
  - Implementation using *XMLHttpRequest*
  - Each notification is implemented as a separate **XHR download pattern, scheduled periodically**

```
function checkUpdates() {
  var xhr = new XMLHttpRequest();

  xhr.open('GET', '/update');

  xhr.onload = function() {
    if (this.status == 200) {
      document.getElementById('update').innerText = this.response;
    }
  };

  xhr.send();
}

setInterval(checkUpdates(), 60000);
```

# Real-Time Notifications on the Web


- Server-To-Client notifications

Example

- Size of notification payload data: 10 bytes
- Typical size of HTTP request header: cca. 500 bytes
- Typical size of HTTP response header: cca. 500 bytes
- Maximum allowed notification delay: 5 seconds
- Longest period between two consecutive server updates: 1 hour
- 10,000 clients connected to a server

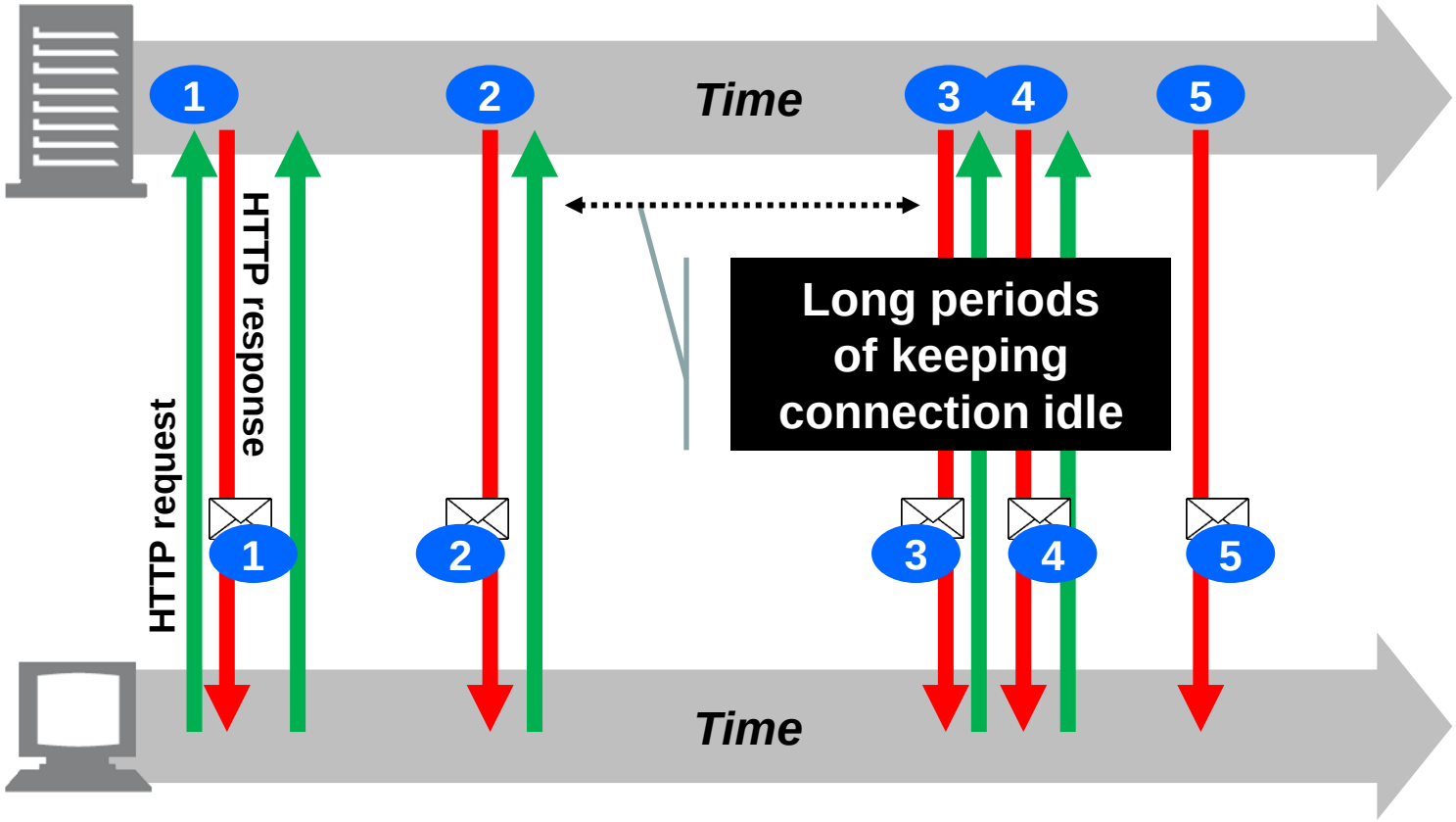
$$\boldsymbol{efficiency\ of\ communication} = \\ = \frac{\mathbf{10}}{\mathbf{719} * (\mathbf{500} + \mathbf{500}) + \mathbf{1} * (\mathbf{500} + \mathbf{500} + \mathbf{10})} \approx \mathbf{0.0013\%}$$

$$\boldsymbol{ideal\ throughput} = \frac{\mathbf{10}*\mathbf{8}}{\mathbf{5}} * \mathbf{10,000} = \mathbf{160\ kbps}$$

$$\boldsymbol{required\ throughput} = \\ = \frac{(\mathbf{719} * (\mathbf{500} + \mathbf{500}) + \mathbf{1} * (\mathbf{500} + \mathbf{500} + \mathbf{10})) * \mathbf{8}}{\mathbf{3600}} * \mathbf{10,000} \approx \mathbf{16\ Mbps}$$

# Real-Time Notifications on the Web

- Server-To-Client notifications
  - Instead of returning empty response, server keeps the connection idle until an update is available ⇒ **LONG POLLING**
  - No delays, but waste of server network resources (TCP connections) is likely to occur (long-lived connections)



# Real-Time Notifications on the Web


- Server-To-Client notifications
  - Implementation using *XMLHttpRequest*
  - Each notification is implemented as a separate **XHR download pattern, scheduled immediately upon completion of a previous one**

```
function checkUpdates() {
  var xhr = new XMLHttpRequest();

  xhr.open('GET', '/update');

  xhr.onload = function() {
    if (this.status == 200) {
      document.getElementById('update').innerText = this.response;
    }
    checkUpdates();
  };

  xhr.send();
}
```

# Real-Time Notifications on the Web


- Server-To-Client notifications

  Example
  - Size of notification payload data: 10 bytes
  - Typical size of HTTP request header: cca. 500 bytes
  - Typical size of HTTP response header: cca. 500 bytes

$$\boldsymbol{efficiency\ of\ communication} =$$

$$= \frac{payload\ size}{request\ header\ size + response\ header\ size + payload\ size}$$

$$= \frac{\mathbf{10}}{\mathbf{500 + 10 + 500}} \approx \mathbf{1\%}$$

- Network load
  - Much more efficient than polling (1 % vs. 0.0013 %)
  - Efficiency compared to XHR-based client-to-server notification
- Server load
  - **Can waste server resources (TCP connections)**


UNIZG-FER 222464

Web Architecture, Protocols, and Services

# Server-Sent Events (SSE)

# Server-Sent Events (SSE)

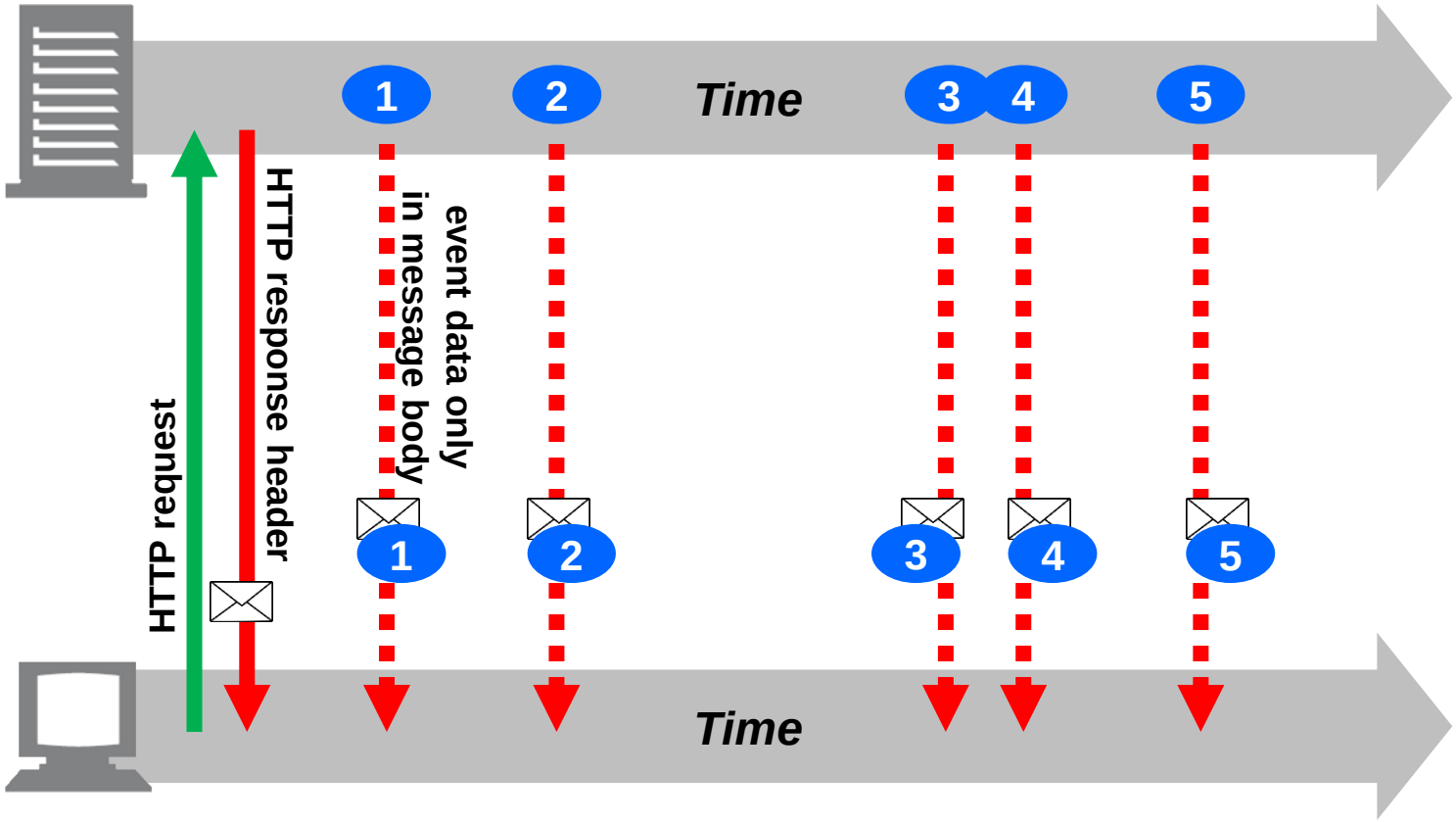

- Enables efficient **server-to-client** streaming of **text-based** event data
- Introduces two new browser API components

| | |
|---|---|
| `EventSource` | Interface which allows the client to receive push notifications from the server as DOM events |
| `event stream` | Data format used to deliver the individual updates |

- Characteristics
  - Low latency delivery via a single, long-lived connection
  - Efficient browser message parsing with no unbounded buffers
  - Automatic tracking of last seen message and auto reconnect
  - Client message notifications as DOM events
  - No need for repeated HTTP requests
    - Only one HTTP request is necessary to initialize the event stream

# Server-Sent Events (SSE)

- Client initiates the connection through HTTP request
- Server confirms the connection with HTTP response and keeps the connection alive
- Subsequent updates are appended to the message body of the HTTP response



# Server-Sent Events (SSE)


- Implementation using *EventSource* API

```
var source = new EventSource("/path/to/stream-url");

source.onopen = function () { ... };

source.onerror = function () { ... };

/* default event handler */
source.onmessage = function (event) {
  log_message(event.id, event.data);
  if (event.id == "CLOSE") {
    source.close();
  }
};

/* event-specific handler */
source.addEventListener("foo", function (event) {
  processFoo(event.data);
});
```

STREAM

# Server-Sent Events (SSE)

- `event stream` protocol

**HTTP request**

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
```

**HTTP response**

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry: 15000

data: First message is a simple string.

data: {"message": "JSON payload"}

event: foo
data: Message of type "foo"

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"
```

API

# Server-Sent Events (SSE)


- Auto-reconnect
  - If the connection is dropped, `EventSource` will automatically reconnect to the server to resume the event stream
- Tracking of the last seen message
  - `EventSource` will automatically advertise the ID of the last seen message, such that the lost messages can be retransmitted

**Stream in progress**

```
retry: 15000

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"

id: 43
data: Lorem ipsum
```

**connection dropped**

**.....**

**15 seconds later**

**HTTP request for reconnection**

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
Last-Event-ID: 43
```

**HTTP response**

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked

id: 44
data: dolor sit amet
```

# Server-Sent Events (SSE)

$$efficiency\ of\ communication = \frac{\sum_{i=1}^{number\ of\ events} event\ size_i}{\begin{array}{c} request\ header\ size + \\ response\ header\ size + \\ \sum_{i=1}^{number\ of\ events} event\ size_i \end{array}}$$

$$\lim_{i \to \infty} efficiency\ of\ communication = 1 \Rightarrow 100\,\%$$

- SSE limitations
  - Server-to-client streaming only
  - Limited to text-based UTF-8 encoded data (*other formats should be encoded as base64 string*)

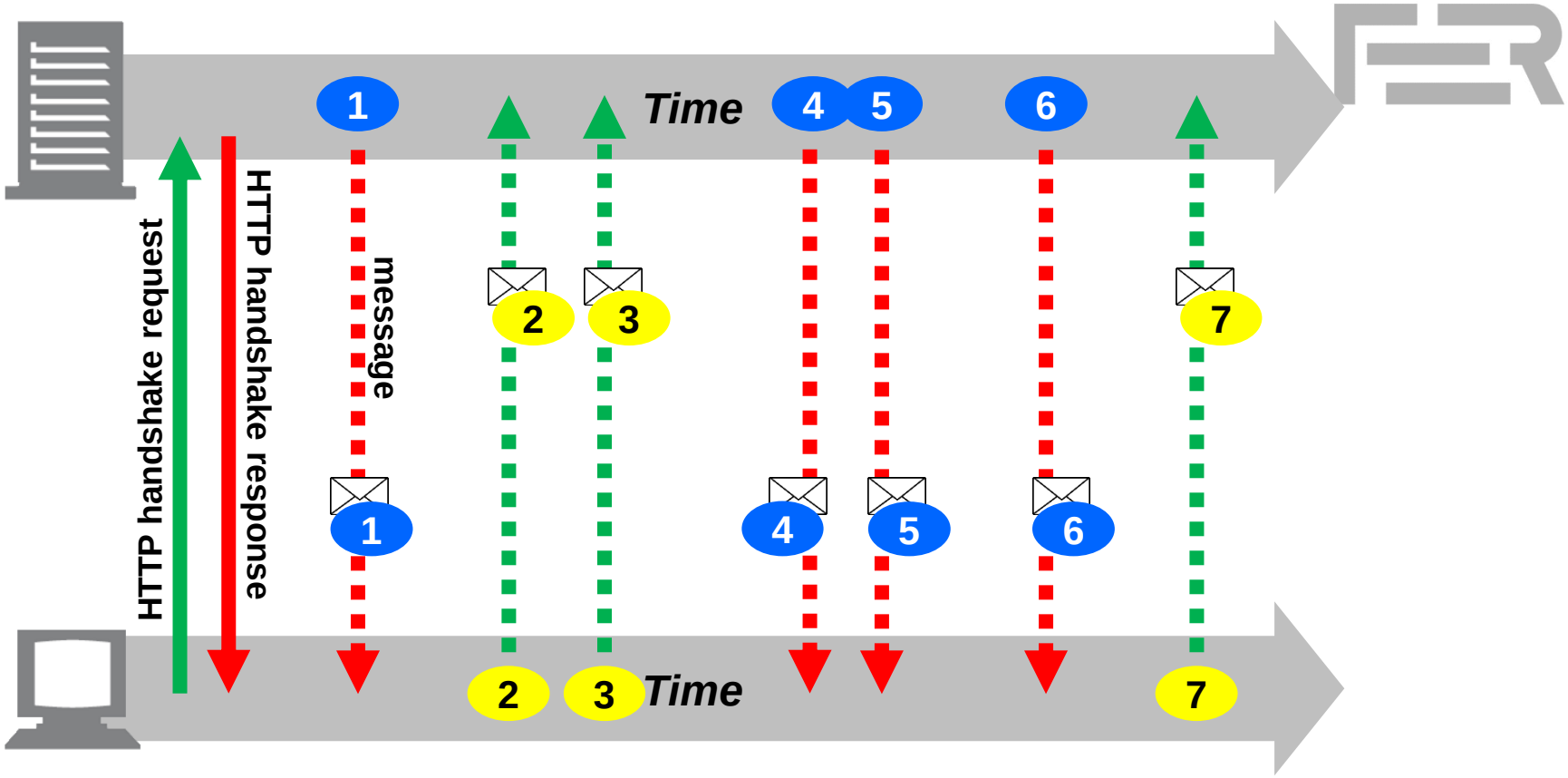

UNIZG-FER 222464

Web Architecture, Protocols, and Services

# WebSocket

# WebSocket

- **Bidirectional**, message-oriented streaming of **text and binary data** between client and server



# WebSocket


- The closest API to a raw network socket in the browser

- Browser (or other library) abstracts away all the complexity behind a simple API and provides a number of additional services:
  - Connection negotiation and same-origin policy enforcement
  - Message-oriented communication and efficient message framing
  - Interoperability with existing HTTP infrastructure

| | |
|---|---|
| `WebSocket protocol` | Network protocol that defines connection management and message framing |
| `WebSocket API` | Programming interface used by web applications (*web browsers, but other client libraries as well*) |

# WebSocket API

```
/* Open a new WebSocket connection */
var ws = new WebSocket('ws://example.com/socket');
```

---

```
/* Optional callback, invoked when the connection is established */
ws.onopen = function () { ... }
```

---

```
/* Optional callback, invoked when the connection is terminated */
ws.onclose = function () { ... }
```

---

```
/* Optional callback, invoked if a connection error has occurred */
ws.onerror = function (error) { ... }
```

---

```
/* A callback function invoked for each new message from the server */
ws.onmessage = function(msg) {
  if (msg.data instanceof Blob) {
    processBlob(msg.data);
  } else {
    processText(msg.data);
  }
}
```

---

```
/* Client-initiated message to the server */
ws.send("Hello server! This is a text message for you.");
```

# WebSocket API

- WebSocket resource URL
  - WebSocket uses its own custom URL scheme (doesn't use `http` or `https`)

| | |
|---|---|
| `ws` | For plain-text communication<br>(e.g. ws://example.com/socket) |
| `wss` | For encrypted communication using SSL/TLS<br>(e.g. wss://example.com/socket) |

- The primary use case for the WebSocket protocol is to provide an **optimized, bidirectional** communication channel between applications running in the browser and the server

- The WebSocket uses a wire protocol other than HTTP, so the URL scheme is changed to reflect that

# WebSocket API

- Sending text and binary data
  - Once a WebSocket connection is established, the client and the server can send and receive text (UTF-8 encoded) and binary messages in both directions over the same TCP connection

```
var ws = new WebSocket('ws://example.com/socket');

ws.onopen = function () {
  /* Sending a text message */
  ws.send("Hello server!");
  ws.send(JSON.stringify({'msg': 'payload'}));

  /* Various ways of sending a binary message.
     Binary options are simply an API convenience: on the wire,
     a WebSocket frame is either marked as binary or text */
  var buffer = new ArrayBuffer(128);
  ws.send(buffer);

  var intview = new Uint32Array(buffer);
  ws.send(intview);

  var blob = new Blob([buffer]);
  ws.send(blob);
}
```

# WebSocket API


- Message ordering
  - The `send()` method is asynchronous: the provided data is queued by the client, and the function returns immediately
  - **Do not mistake the fast return for a signal that the data has been sent**
  - All WebSocket messages are delivered in the exact order in which they are queued by the client
  - As a result, a large backlog of queued messages, or even a single large message, will delay delivery of messages queued behind it – **head-of-line blocking**

# WebSocket Protocol


- Latest version (v13) defined in RFC 6455 (December 2011)
- Two main components:

| | |
|---|---|
| `Connection opening handshake` | used to negotiate the parameters of the connection |
| `Binary message framing mechanism` | allows for low overhead, message-based delivery of both text and binary data |

# WebSocket Protocol


- Connection opening handshake (*HTTP Upgrade*)
  - Requires one HTTP round trip between client and server
  - HTTP is chosen to stay compatible with the existing web architecture

```
var ws = new WebSocket('ws://thirdparty.com/web-socket');
```

**HTTP handshake request**

```
GET /web-socket HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
```

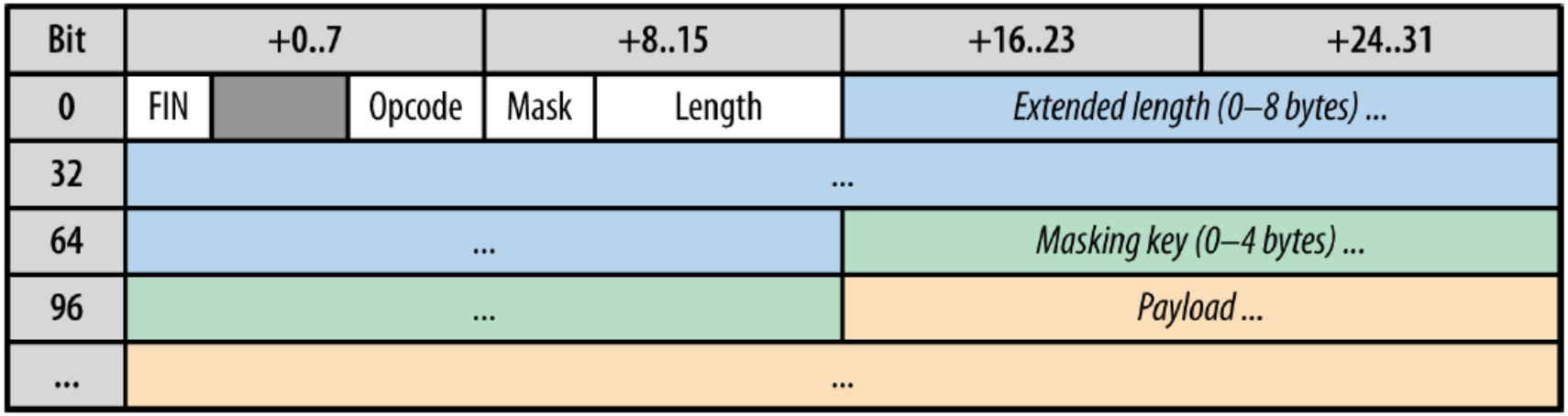
**HTTP handshake response**

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Access-Control-Allow-Origin: http://example.com
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

# WebSocket Protocol

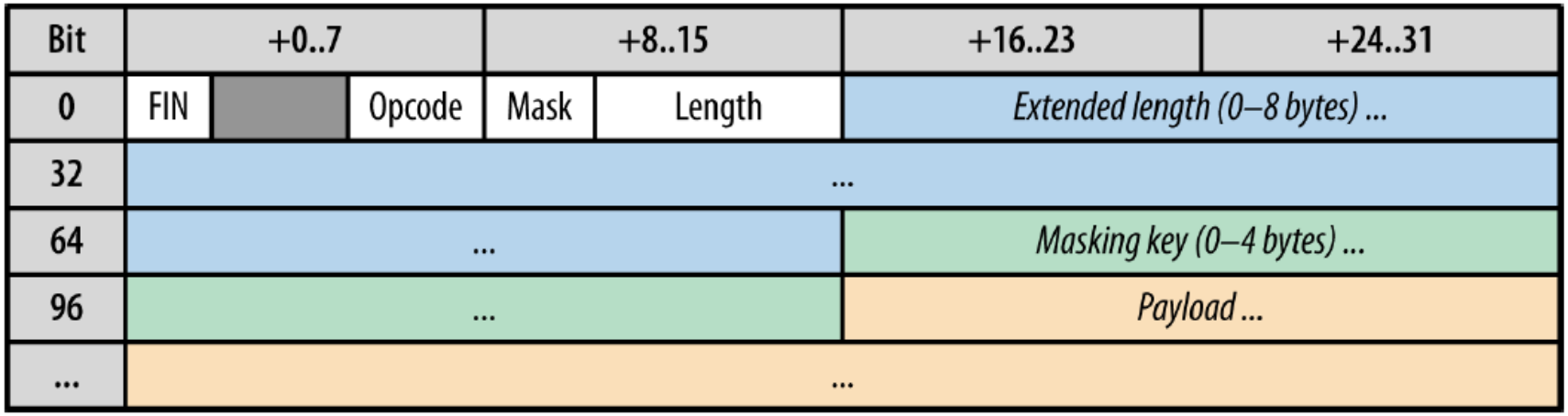
- Binary message framing mechanism
  - From this point on, the HTTP-based communication is finished and all further communication is based on *Websocket frames* (binary protocol)

- WebSocket frame
  - `FIN` bit indicates whether the frame is a final fragment of a message. A message may be transfered as a single frame or split into multiple frames
  - `Opcode` indicates type of transferred frame: text (1) or binary (2) for transferring application data;  or connection close (8), ping (9), and pong (10) for control frames used for connection liveness checks



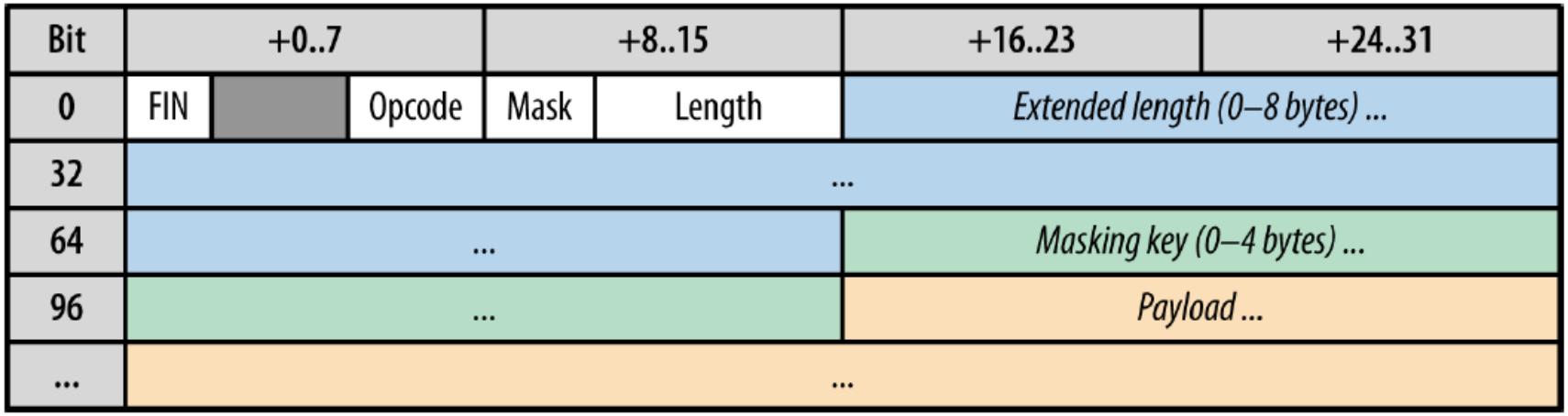
# WebSocket Protocol

- WebSocket frame
  - `Mask` bit indicates whether the payload is masked (for messages sent from the client to the server only, for intermediaries that do not understand the WebSocket protocol)
  - Payload length is represented as a variable-length field
    - If 0–125, then that is the payload length
    - If 126, then the following 2 bytes represent a 16-bit unsigned integer indicating the payload length
    - If 127, then the following 8 bytes represent a 64-bit unsigned integer indicating the payload length



# WebSocket Protocol

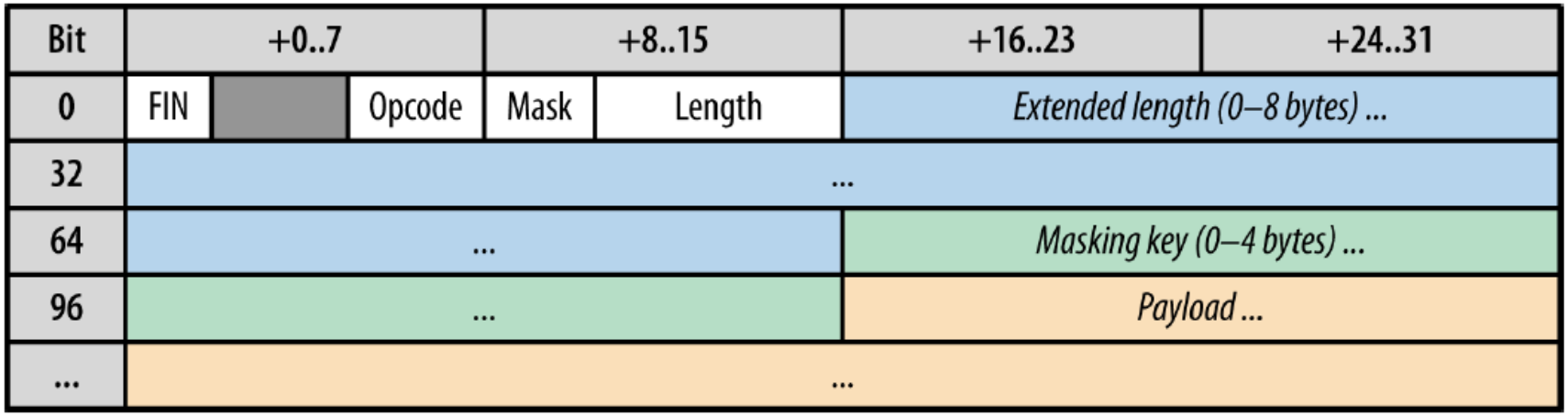
- WebSocket frame
  - `Masking key` contains a 32-bit value used to mask the payload
    - XOR operation is applied to mask 32-bit blocks of payload
  - `Payload` contains the application data



# WebSocket Protocol

- Framing overhead
  - Due to variable `Length` field, framing overhead is minimized

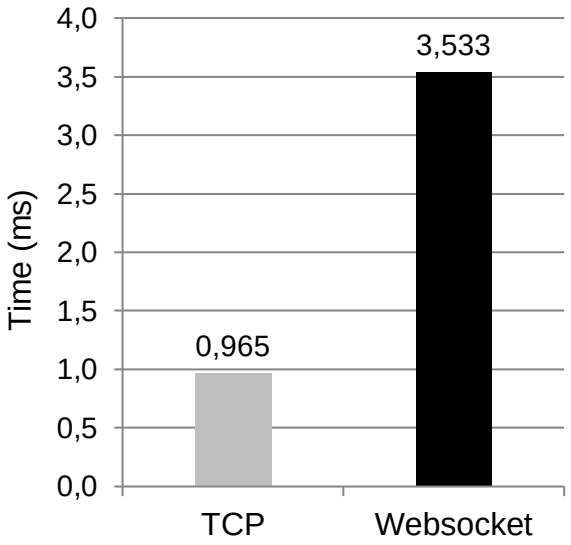
| Message length (in bytes) | Client message | Server message |
|---|---|---|
| `up to 125` | 6 | 2 |
| `126 to 64k` | 8 | 4 |
| `over 64k` | 14 | 10 |



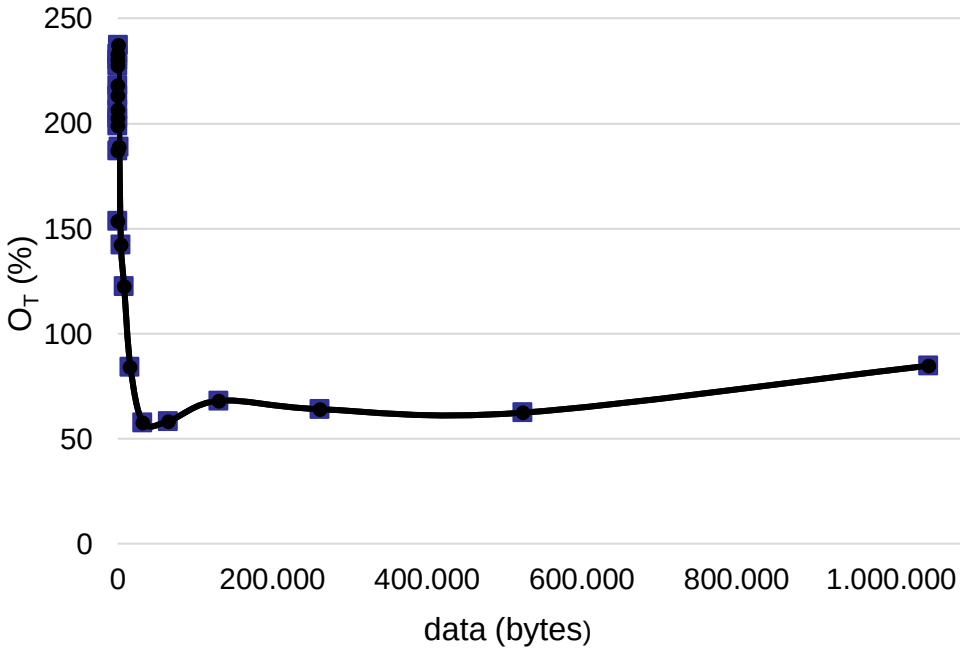
# WebSocket Protocol

- Websocket performance related to plain TCP

**Connection handshake duration (*HTTP Upgrade*)**




**Overhead in message transfer duration ($T_{WebSocket}$ / $T_{TCP}$)**




**Source:**
D. Skvorc, M. Horvat, S. Srbljic: **Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams**, 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014

# Summary of XHR vs. SSE vs. WebSocket


- Each method has its own area of applicability
  - XHR
    - Simple for transactional data transfers (HTTP), not a good choice for data streaming
  - SSE
    - Efficient for streaming, but unidirectional and limited to text-only data
    - Requires long-lived connections
  - WebSocket
    - Most efficient and flexible, but still requires long-lived connections

| | XMLHttpRequest | Server-Sent Events | WebSocket |
|---|---|---|---|
| Request streaming | no | no | yes |
| Response streaming | limited | yes | yes |
| Framing mechanism | HTTP | event stream | binary framing |
| Binary data transfers | yes | no (base64) | yes |
| Compression | yes | yes | limited |
| Application transport protocol | HTTP | HTTP | WebSocket |
| Network transport protocol | TCP | TCP | TCP |